Machine Learning Assignment 1

Author: Alex O Regan

Year 4

Class: SDH4-B

# Table of Contents

Basic Evaluation ………………………………………………………………………... 2

Research and Detailed Evaluation ………………………..……………………………... 4

Stopwords ……………………..……………………………..………………….. 4

HTML Entities ……………………………..……………………………………….… 5

Emojis ……………………..……………………………..………………….. 5

Negation ……………………..……………………………..………………….. 6

Lemmatization ……………………..……………………………..………………….. 7

Conclusion ……………………………………………………………………….. 7

# Basic Evaluation

### Cleaning the DataSet

For cleaning the training / testing datasets, I primarily used two libraries: Python Regular Expressions (package ‘re’), and Pandas. The actual sanitization of the data was broken up into two sections - necessary sanitization and extra sanitization. Necessary sanitization is any processing that is essential to execute before the data is useable, while extra sanitization includes operations such as the removal of special characters. The necessary sanitization is as follows:

1. I first specified to Pandas to only read the first 4 columns in any CSV I gave it - since it is intended for the CSVs to only have 4 columns, any extra columns it found would surely represent a corruption of the file. This step was necessary, because otherwise Pandas would become confused by the mismatching columns and crash.



1. The second piece of sanitization done on the data was to drop any records from the document that were empty. The training CSV contained a number of ;’s at the bottom of the file, that Pandas mistakenly reads as rows. Dropping these empty rows only required a single function call.



1. The final necessary sanitization done was the conversion of the sentiment column values from floats to integers. For some reason, Pandas read the sentiment column as floats (i.e 1.0 , 0.0), which made boolean operations more difficult. I cast these values to integers 0 and 1.



The above operations allowed me to work with the data without fear of encountering unusable data. However, further processing needed to be done to discard anything that might be ‘junk’.

1. I first made all characters read lowercase. This seems like a logical thing to do, since case is very rarely associated with sentiment, unless someone is shouting in all-capitals, but even then, sentiment isn’t guaranteed. The addition of the lowercase check improved the model’s accuracy by 1.5%.
2. The second check done removes any special characters from the text, based on a regular expression. Characters can be taken out of the regular expression in order to keep them in the final words, but most characters didn’t seem to provide any improvement to performance, so the regular expression removes nearly all of them. There is a case to be made for leaving ‘:’ , ‘(‘, ‘)’ , etc in the text, since these characters represent emojis, however, this will also result in non-emoji uses of the characters showing up. In the end, filtering out the emoji characters diminished performance by 0.7%, but this can be remedied later on by doing a preliminary check for emojis specifically.
3. Finally, any spaces that were made up of more than one whitespace character were replaced with a single whitespace. This doesn’t have any effect whatsoever on performance, but helps with human-readability.

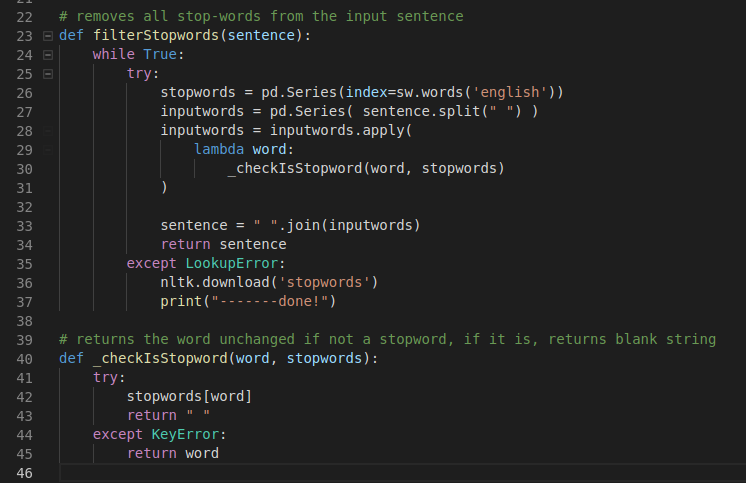
The addition of these extra sanitization steps boosted accuracy of the model from 73.09% to 75.58%.

# Research and Detailed Evaluation

Before the addition of any further preprocessing, the model’s accuracy stood at 75.58%.

### Stopwords

The first and arguably easiest preprocessing to implement was the removal of stop-words. I did this by cross-checking each tweet with the NLTK package’s list of stopwords.



After implementing this functionality, the accuracy sadly dropped. I didn’t expect this change to impact performance very much, as I thought there wouldn’t be many stopwords present in Tweets that were spelled correctly, but as it turned out, it filtered enough stopwords to decrease the accuracy from 75.58% to 72.94%.

In general, one might not expect there to be a relationship between sentiment and stopwords, but in the provided dataset, there was enough of a correlation between the two to impact the accuracy significantly.

### 

### HTML Entities

While it is quite a trivial concept to remove special characters from the input sentences, one type of special character I had not considered was the HTML Entity. These are special characters that are purposely written using other non-special symbols. For example, the ‘&’ character as a HTML Entity would be written as ‘&amp;’. I noticed that there were quite a few of these scattered around the Tweets, so I added them to my special characters filter, and it allowed my accuracy to rise from 72.94% back up to 73.38%.



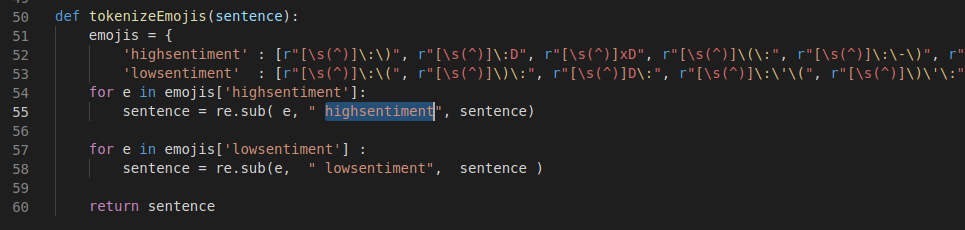
### Emojis / Emoticons

The concept of using emojis as a means of analysing sentiment seemed like quite a good idea, since you would expect a Tweet to have a positive sentiment if it contained a smiling emoji. That being said, there were a few challenges in implementing this.

Primarily, this would be the first function that might involve directly altering P(word|class) values, and due to my preprocessing function being executed before calculating P(word|class), I would have to call the emoji function in a different location to the rest of my filters. This would be possible, but would also lead to slightly messier code. On-top of this, I would also have to determine an optimal P(word|class) value for each emoji, which again, while possible, might not be practical because it could turn out to be too much work in the time I had available.

With these reasons in mind, I first tried a different solution.

Instead of directly inserting emoji P(word|class) values, I would simply tokenize any emojis found. I replaced positive/happy emojis with the word “highsentiment”, and replaced sad/negative emojis with the word “lowsentiment”.

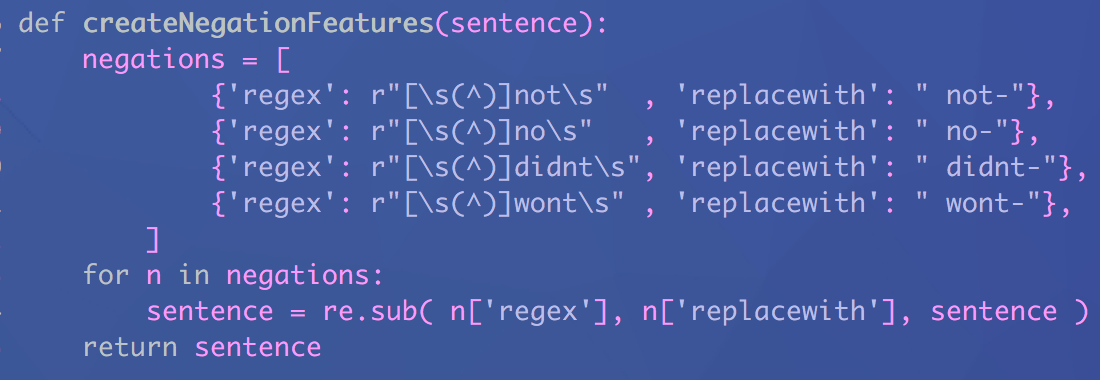


The emojis in the code above were written as regular expression, so that I could better-control what was considered an emoji. The above regex makes sure there aren’t letters before emojis, such as in the case of “ (username): ”, where a “ ): ” would normally be detected.

This solution had the benefit of being easier to work into my existing code, but the main problem I see with it is that it is difficult/impossible to determine exactly how much these words will affect a sentence’s sentiment. Replacing the word ‘highsentiment’ with the word ‘goodsentiment’ might lead to unexpected changes in accuracy due to unpredictable training sets, and words having different P(word|class) values. This approach ended up lowering the model’s accuracy by a further 0.14%, however I believe this is due to seemingly contradictory tweets such as “ I’m well ): ”, which will of-course reduce the effectiveness of the Bayes model. If I were to add these same emojis directly into my P(word|class) table, I would expect to have the same issues, due to these contradictory tweets.

### Negation

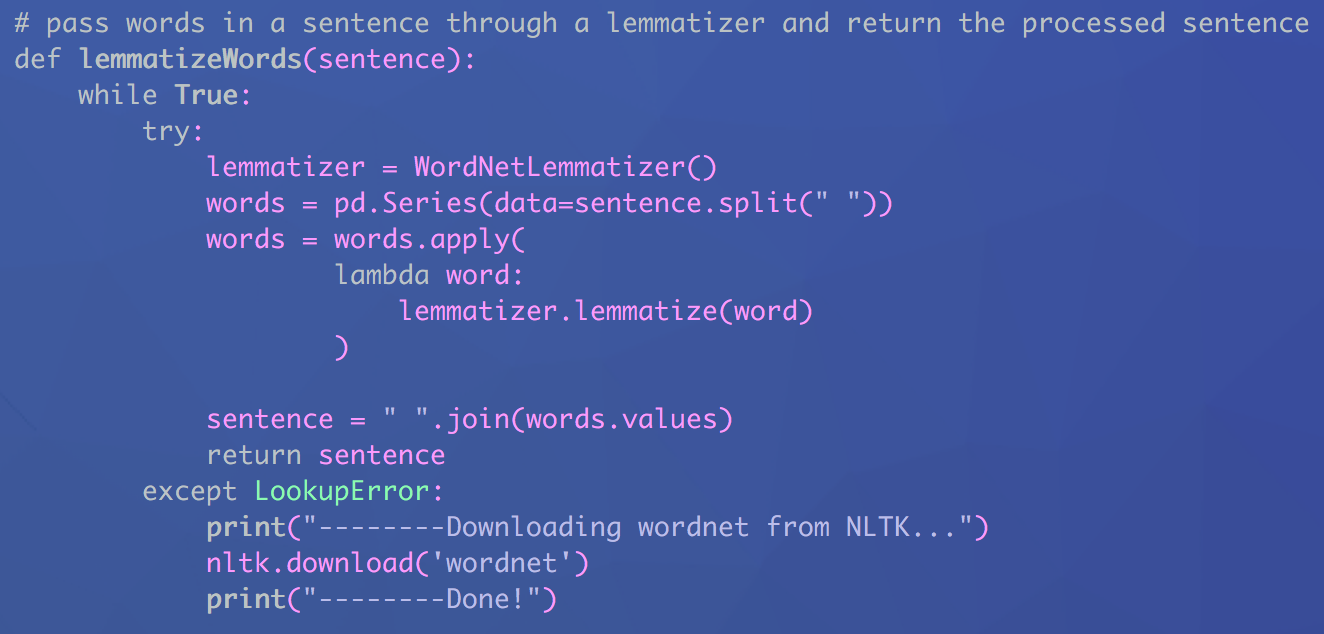
The next approach to preprocessing I attempted was to find words like ‘not’ , ‘didn’t’, ‘won’t, etc, and join them to the following word ( “not good” becomes “not-good” ). I had to be careful of the order that I executed my preprocessing functions in , since hyphens I added in (i.e “not-good”) would be removed by the special characters filter if it was called after negation. Prematurely removing stopwords could also potentially affect the negation filter’s performance.



After accounting for these quirks, the addition of negated words improved the accuracy of the model from 73.24% to 73.31% ( .7% ).

### Lemmatization

NLTK package provided an easy-to-use lemmatizer function that I incorporated into my project. I hoped lemmatization would increase the effectiveness of the dataset by boiling all words down to their base word, allowing the classifier to extract more meaning from sentences that contain variations of the same word.



In the end the lemmatizer increased my classifier accuracy by .44% , which wasn’t much, but still represented a positive modification to the dataset.

# Conclusion

The impacts of each type of preprocessing were interesting to investigate, as it was possible to see how each process would affect the analysis given how chaotic and mangled the Twitter data originally was. In future, I’d like to be able to apply similar operations to more coherent data, to see how the changes impact the final accuracy of sentences that make more sense.

With every preprocessing function enabled, my model’s final accuracy was 73.75%, how ever if I cherry-picked which preprocessing to perform, I was able to achieve up to 75.65%. This was achieved by only filtering blank spaces and special-characters / HTML-entities. Again, I believe this increase in accuracy when doing less preprocessing is due to how messy and chaotic the data is.